MAST 680 Assignment 2

Serly Ishkhanian

March7, 2023

Abstract

In this paper, we will analyze the video recordings of a spring-mass system from three different angles to extract the equation of motion of the mass in two separate cases: the ideal and the noisy case. In both cases, we use the Frobenius norm to track the motion of the mass across the frames of each of the videos; this gives us a six-dimensional dataset. To reduce the dimensionality so that the motion is in 2D, we use Principal Component Analysis (PCA). Then, we use the Sparse Identification of Nonlinear Dynamics (SINDy) method to learn the equations of motion that govern the dynamics of this system.

1 Introduction and Overview

Knowing the equations of motion that govern the dynamics of a system can be crucial since they describe how the system behaves over time and assists in predicting its behavior. The Sparse Identification of Nonlinear Dynamics (SINDy) method is a data-driven technique that attempts to discover the equations of motion of a dynamical system [2]. This method provides a way of describing the data with nonlinear models using the sparse regression technique to determine the significant terms in ordinary differential equations.

Given six videos created from three different cameras of a paint can attached to a spring, this paper aims to describe the equation of motion of the paint can in the ideal and noisy cases. In the ideal case, the paint can moves in the z direction with a simple harmonic motion. The noisy case is similar to the ideal case. However, the three videos include camera shakes which could make it difficult to extract the simple harmonic motion. For each case, we will start by tracking the position of the paint can across the frames in each of the videos using the Frobenius norm, giving us a six-dimensional dataset corresponding to the x and y coordinates of the paint can in each of the three videos. Since motion is one-dimensional, we will perform PCA to reduce the dimension and remove the correlation among the rows. Lastly, we will apply the SINDy method to learn the equations of motion underlying this dynamical system using the pysindy library [4] [5] on python. Theoretical backgrounds of the techniques used are in Section 2, and implementation and results of those techniques are in Sections 3 and 4, respectively.

2 Theoretical Background

2.1 Frobenius Norm

To construct the equation of motion of the spring-mass system, first, we need to extract the mass positions from the video frames. So, how do we track the paint can? To do so, we will use the Frobenious norm $\|\cdot\|_F$, which is the square root of the sum of the squares of the elements of a matrix. Interestingly, if we want to measure the closeness of two matrices A and B, then analyzing the value of $\|A-B\|_F$ suffices.

How does measuring the closeness of two matrices help us track the position of the paint can? Let's recall that a video is a sequence of frames where the frames are either three-dimensional (colored) or two-dimensional (grayscale) data matrices. Now, imagine we have a picture of the

first frame of one of the videos and a picture of the paint can that we want to track. Place the reference picture over the frame and move it along the frame until we find a match. By observing the pictures, we know we located the can in the frame, correct? In the matrix world, as mentioned, a gray picture can be represented by a two-dimensional data matrix. Numerically, accomplishing the same process as above implies computing the Frobenius norm of the difference of the matrices corresponding to the reference picture and the part of the frame we are comparing it with. If it is a match, then this Frobenius norm will have the smallest value compared to all the others.

Once we find a match, we calculate the center of the rectangular region. For camera 1, we will denote the coordinates of the center as (x_a, y_a) at each instance in time. Then, making vectors of the positions at each time and repeating this process for cameras 2 and 3, we obtain the data matrix $X \in \mathbb{R}^{6\times n}$ where n represents the minimum number of frames from the three videos,

$$X = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{y}_a \\ \mathbf{x}_b \\ \mathbf{y}_b \\ \mathbf{x}_c \\ \mathbf{y}_c \end{bmatrix}.$$

2.2 Principal Component Analysis (PCA)

When the rows or the columns of a matrix (the variables in the experiment) are dependent, we will have redundancies. To remove those redundancies, we perform PCA. PCA will project the data onto a lower dimensional space where the principal directions retain the maximum variation of the original dataset, keeping information loss at a minimum and causing the variables (in the lower dimension) to be uncorrelated.

Since PCA projects the data onto directions containing the maximum variance, if the data is not centered, then this would affect the result because if one variable has a larger variance than the others, PCA will get skewed towards that direction. Thus, first, we center the data. Let $\bar{x}_i$ denote the row means of X. So, the matrix containing all the means $\bar{X} \in \mathbb{R}^{n\times 6}$ is:

$$\bar{X} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} [\bar{x}_1 \bar{x}_2 \cdots \bar{x}_6]. \tag{1}$$

The mean-centered data is:

$$X_{\mathrm{c}} = X - \bar{X}^T. \tag{2}$$

Now, to reduce the dimensions of our data, we apply PCA. We have two approaches; either compute the covariance matrix of X_c and observe its eigendecomposition, or obtain the Singular Value Decomposition (SVD) of X_c. PCA is the SVD of mean-subtracted data [2]. In this paper, we use the SVD of the mean-centered data matrix, $X_c = U\Sigma V^T$, to perform dimensionality reduction. Where U, V are unitary matrices such that $U^{-1} = U^T$, $V^{-1} = V^T$, and Σ is a diagonal matrix whose entries are the ordered singular values of X_c [1]. Furthermore, columns of $U \in \mathbb{R}^{6\times 6}$ represent the principal directions of X_c, columns of $V \in \mathbb{R}^{n\times n}$ represent the temporal evolutions of the data, and the singular values retain significant information regarding our data. In addition, the squared singular values represent the variances of the variables (which are also the eigenvalues of the covariance matrix). For more details on the connection between PCA and the SVD, please refer to [2].

Using the first s columns of U, $U_s \in \mathbb{R}^{6\times s}$, we project X_c onto the directions with maximum variance (i.e. data gets projected onto a subspace spanned by the columns of U_s where the new variables are uncorrelated) by computing

$$X_p = U_s^T X_c \in \mathbb{R}^{s\times n}. \tag{3}$$

2.3 Sparse Identification of Nonlinear Dynamics (SINDy)

The SINDy method is a data-driven technique that attempts to identify the governing equations of a dynamical system i.e. it aims to express the system as a nonlinear dynamical system [3]:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)), \tag{4}$$

where $\mathbf{x}(t) \in \mathbb{R}^m$ represents the state of the system at time t, and $\mathbf{f}(\mathbf{x}(t))$ is the nonlinear function representing that define the equations of motion of the system. Given the data matrix $X \in \mathbb{R}^{n\times m}$ (in our case, $X = X_p$),

$$X = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_m(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_m(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_n) & x_2(t_n) & \cdots & x_m(t_n) \end{bmatrix}, \tag{5}$$

the aim is to determine f. What SINDy does is that it considers a dictionary of functions,

$$\mathcal{D} = \{\theta_1\theta_2\cdots\theta_K\}, \quad K \geq 1 \tag{6}$$

where $\theta_k : \mathbb{R}^m \to \mathbb{R}$ for each $1 \leq k \leq K$. Then, we evaluate the data at each of the K functions of the dictionary, and obtain the matrix

$$\Theta(X) = \begin{bmatrix} \Theta_1(X_1) & \Theta_1(X_2) & \cdots & \Theta_1(X_m) \\ \Theta_2(X_1) & \Theta_2(X_2) & \cdots & \Theta_2(X_m) \\ \vdots & \vdots & \ddots & \vdots \\ \Theta_K(X_1) & \Theta_K(X_2) & \cdots & \Theta_K(X_m) \end{bmatrix} \in \mathbb{R}^{K\times m}, \tag{7}$$

which is a library of nonlinear functions of the states. Moreover, SINDy assumes that only a few of these nonlinearities are active. Hence, sparse regression problem is set up to identify the sparse vectors of coefficients $\Xi = [\xi_1\xi_2\cdots\xi_m]$ that determine those active nonlinearities:

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi. \tag{8}$$

To find those sparse vector of coefficients ξ_k for the k^{th} row equation of (8), an optimization problem is solved. Moreover, when estimating the differential equations, small corruptions in the data or estimations of the numerical derivative could result in extra small nonzero coefficients in the obtained model. To avoid these issues, SINDy defines a sparsity parameter $\lambda > 0$ and assigns zero coefficients to all the entries in the dictionary corresponding to an absolute value of a coefficient in Ξ that is smaller than λ [1]. By repeating this procedure, SINDy can shorten the size of the dictionary by eliminating the nonlinear functions with coefficients that fall below the threshold λ. Thus, SINDy is able to estimate the model representing the nonlinear dynamics underlying the data matrix.

3 Algorithm Implementation and Development

Before implementing the above techniques, we must load the MATLAB data files and convert them into data files that python can work with. Each video data has a shape (frame height, frame width, channels, number of frames). If the value of the channels is 3, then we have coloured video. Furthermore, we convert the data into grayscale data, refer to Appendix B, since python cannot compute the Frobenius norm of a three-dimensional array.

3.1 Tracking the Mass

First, let's discuss the ideal case. For each video data, we start by locating the region corresponding to the paint can described as a 2D matrix $C_i \in \mathbb{R}^{h\times w}$ for $i = 1, 2, 3$. This will be used

as the reference picture. Define two vectors for the x and y coordinates of the center. Then, we do the following for each video data:

From Algorithm 1, we obtain the matrices $X \in \mathbb{R}^{6\times 226}$, $X_{\text{noisy}} \in \mathbb{R}^{6\times 314}$ for the ideal case, and the noisy case, respectively. For more details, please view the code in Appendix B. To cut down computation time, regions of interests were specified to seach for the can in (since the can only moves up and down). Lastly, this process is repeated with noisy video data using the same reference regions used in the ideal case.

Algorithm 1: Track the Position of the Mass

for $i = 1 : n$(no. of frames) **do**
  Set a large arbitrary score value = 10000000.
  Take a submatrix $TC \in \mathbb{R}^{h\times w}$ from the frame, and move along the frame.
  Compute $\|C_i - TC\|_F$
  if $\|C_i - TC\|_F <$ value **then**
    value = $\|C_i - TC\|_F$
    Let x' and y' denote the upper left corner of TC
  end if
  Append to the the coordinate vectors $x = \frac{x'+x'+w}{2}$, and $y = \frac{y'+y'+h}{2}$, respectively.
end for
Repeat this for the three videos.
Create matrix X as mentioned in Section 2.1.

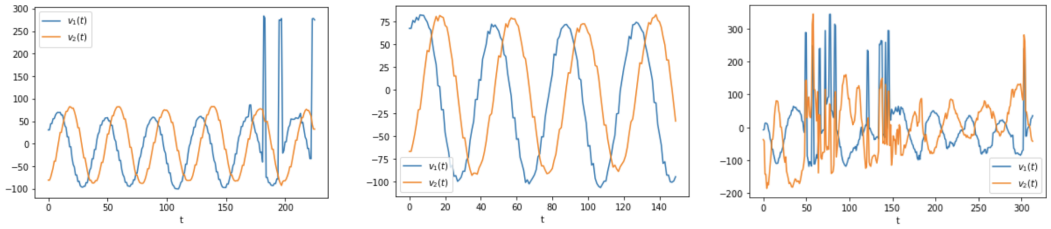
3.2 Dimensionality Reduction

Once we obtain the matrix X, clearly, we observe that these data are in six dimensions, even though motion can be described in one dimension. To remove the redundancies along the rows, and reduce the dimensionality, first, we center the data. As mentioned in Section 2.2, we compute the row means and create the matrix $\bar{X}$. We obtain the mean-centered data by computing $X_c = X - \bar{X}^T$. The reason why we are taking the transpose is because $\bar{X} \in \mathbb{R}^{226\times 6}$, $\bar{X} \in \mathbb{R}^{314\times 6}$ for the ideal case and noisy case, respectively.

Next, we compute the SVD of X_c. We observe from Figure 1a that the curves are not behaving the way they should be behaving after $t = 150$. So, we truncate X_c until the first 150 columns, $X_c' \in \mathbb{R}^{6\times 150}$. After truncating, we observe the behaviour of the temporal evolutions until $t = 150$ in Figure 1b. Notice how different the behavior of the temporal evolutions are for the noisy data in Figure 1c, which is expected since the data is noisy. Moreover, Figure 2 demonstrates the energy of each of the squared singular values σ_i^2, energy $= \frac{\sigma_i^2}{\sum_{i=1}^{6}\sigma_i^2}$ for the truncated data in the ideal case, and the data of noisy case (not truncated). Although it was mentioned that motion is one dimensional, we can see in Figure 2a that we have two dominant nonzero singular values, with cumulative energy of 96.88%. In Figure 2b, the two dominant singular values amount to 86.10% of cumulative energy. Hence, we will project both datasets onto two dimensions by computing $X_p = U_2^* X_{\text{centered}}$. First singular value corresponds to the position, while the second corresponds to the velocity of the mass. Lastly, we observe how the remaining singular values (apart from the first two) are nonzero in Figure 2b, unlike the singular values in the ideal case. This can be caused by the existence of noise in the data.

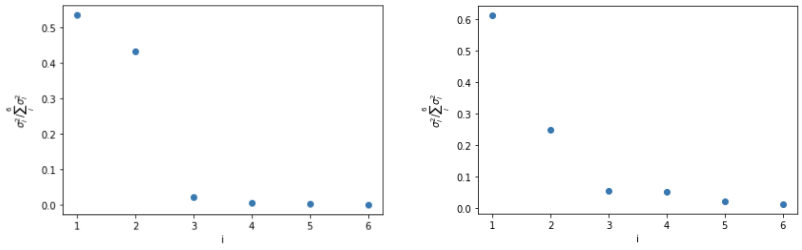
3.3 The SINDy Method

Now that we have our projected data, we will apply the SINDy method to obtain the equations of motion. To do so, we use the python library pysindy [4] [5]. We set the value of the sparsity parameter (the threshold) to be $\lambda = 0.1$, and the time step t between each data point to be $t = 1/$frames per second. For more details on what the following functions in Algorithm 2 are



(a) (b) (c)

Figure 1: Plot of the temporal evolution of the first two observations of the (a) ideal case using X_c, (b) ideal case using X_c', and (c) noisy case using X_c corresponding to the noisy data.



(a) Ideal Case (b) Noisy Case

Figure 2: Plots of the energy ratios of the squared singular values for both ideal and noisy cases.

Algorithm 2: Apply the SINDy Method

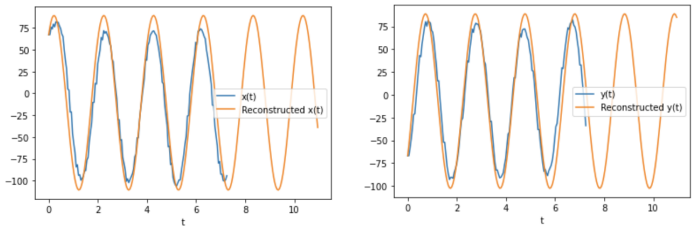
Define the feature names corresponding to the variables in your data
Feature_Names = ["x","y"]
Specify the optimizer and threshold
Optimizer = pysindy.STLSQ(threshold = 0.1)
Define the model
model = pysindy.SINDy(feature_names = Feature_Names, optimizer = Optimizer)
Fit the model
model.fit(X_p, t= duration of video/number of frames)
Print the model
model.print()

responsible for, refer to Appendix A. Algorithm 2 estimates the differential equations $\dot{x}(t)$ and $\dot{y}(t)$ that govern the dynamics of the system. To find $x(t)$ and $y(t)$, we will solve the ordinary differential equations by using the python function **odeint** using two initial conditions obtained from the first row of the projected dataset. We choose the first row since it corresponds to the values of $x(t)$ and $y(t)$ at $t = 0$.

4 Computational Results

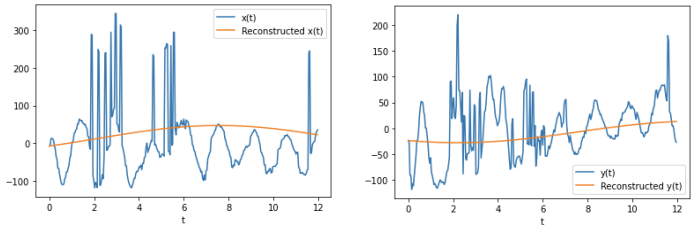
In this section, we will present the results obtained from implementing the algorithm in Appendix B in Python to obtain the equations of motion that govern the dynamics of our system.

By applying the SINDy method to the projected data, we obtained the following estimated differential equations presented in Table 1 for both the ideal case and the noisy case using $\lambda = 0.1$ and $\lambda = 0.01$. Comparing the estimation resulting from $\lambda = 0.1$ and $\lambda = 0.01$, notice that the equations from $\lambda = 0.01$ have extra terms. As λ increases, those extra terms disappear. Since the temporal evolutions have graphs of sines and cosines, we expect the differential equation of x to have a term in y and the differential equation of y to have a term in x because integrating these would provide result in $x(t)$ and $y(t)$ to be in terms of sines ans cosines. Thus,



(a) (b)

Figure 3: Plots of $x(t)$ and $y(t)$ with the estimated $x(t)$ and $y(t)$ obtained by integrating the system of differential equations of the ideal case with $\lambda = 0.1$.



(a) (b)

Figure 4: Plots of $x(t)$ and $y(t)$ with the estimated $x(t)$ and $y(t)$ obtained by integrating the system of differential equations of the noisy case with $\lambda = 0.1$.

to proceed, we use $\lambda = 0.1$. Then, by integrating the system of differential equations on the intervals [0, duration of the shortest video], using initial conditions (at $t = 0$) as the first row of X_p, we compute $x(t)$ and $y(t)$ using the function **odeint**. Figures 3 and 4 illustrate the plots of the original $x(t)$ and $y(t)$ with their estimations. We observe that the results of the ideal case are more accurate than the results of the noisy case. Although it seems that there is a difference in amplitudes in Figures 3a and 3b, overall, the estimates seem to be successful. While for the noisy case, the results are not successful.

Table 1: Estimations of the differential equations obtained from applying the SINDy method.

Sparsity Parameter (λ)	Ideal Case	Noisy Case
0.1	$\begin{cases} \dot{x} = -22.253 - 3.237y \\ \dot{y} = 32.032 + 2.969x \end{cases}$	$\begin{cases} \dot{x} = 3.2851 - 0.475y \\ \dot{y} = 2.4061 + 0.181x \end{cases}$
0.01	$\begin{cases} \dot{x} = 69.529 - 0.286x - 3.387y - 0.011x^2 - 0.014y^2 \\ \dot{y} = 139.2031 + 2.589x - 0.204y - 0.015x^2 - 0.014y^2 \end{cases}$	$\begin{cases} \dot{x} = 3.285 - 0.475y \\ \dot{y} = 2.406 + 0.181x - 0.018y \end{cases}$

5 Summary and Conclusions

In this report, we discussed the theoretical background of the methods used to obtain the equations of motion of a paint can hanging from a spring as it moves up and down in simple harmonic motion considering three different videos under the ideal case and the noisy case, which included camera shakes. In Section 3.1, we viewed the algorithmic process of utilizing the Frobenius norm to track the mass positions throughout the frames. After obtaining a six-dimensional data matrix X in both cases, we centered the data and applied SVD to project the data onto the two directions with the maximum variance (where the first and second new variables represent the position and velocity of the mass, respectively). Furthermore, using the projected data, we use the pysindy library on python to estimate the differential equations underlying the system using sparsity parameter $\lambda = 0.01$. Lastly, we use the function **odeint** in python to solve the system of differential equations to visualize the estimated functions $x(t)$ and $y(t)$ in Section 4.

References

[1] Jason J. Bramburger. *Data-Driven Methods for Dynamical Systems.* Concordia University, 2023.

[2] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control.* Cambridge University Press, 2022.

[3] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the national academy of sciences*, 113(15):3932–3937, 2016.

[4] Brian de Silva, Kathleen Champion, Markus Quade, Jean-Christophe Loiseau, J. Kutz, and Steven Brunton. Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, 5(49):2104, 2020.

[5] Alan A. Kaptanoglu, Brian M. de Silva, Urban Fasel, Kadierdan Kaheman, Andy J. Goldschmidt, Jared Callaham, Charles B. Delahunt, Zachary G. Nicolaou, Kathleen Champion, Jean-Christophe Loiseau, J. Nathan Kutz, and Steven L. Brunton. Pysindy: A comprehensive python package for robust sparse system identification. *Journal of Open Source Software*, 7(69):3994, 2022.

Appendix A Python Functions

- `scipy.io.loadmat` Loads MATLAB file
- `np.mean( array, axis = specify axis)` Computes the mean along the specified axis.
- `np.linalg.norm(A,'fro')` Compute the Frobenius norm of matrix A.
- `plt.imshow` Display data as an image.
- `savetxt` Save an array to a text file.
- `loadtxt` Loads data from a text file.
- `np.linalg.svd(A)` Computes the SVD of the matrix A.
- `pysindy.STLSQ(threshold=` λ`)` Sequentially Thresholded Least Squares Algorithm optimizer method used in pysindy to solve for Ξ.
- `pysindy.SINDy(feature_names = , optimizer = )` builds a SINDy model. The argument feature_names is used so that the model prints out the correct labels for x and y.
- `model.fit(data, time step)` Fits the model to the data with the time step between data points specified.
- `model.print()` Prints out the differential equations.
- `odeint( function,` z_0`, t)` Integrates a system of ordinary differential equations (function), with initial condition z_0 specified, and time step.

Appendix B Python Code

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import scipy.io
```

```
from numpy import savetxt
from numpy import loadtext

"""# TRACK THE CAN"""
"""# Load data wit no noise (IDEAL CASE)"""

c1 = scipy.io.loadmat("cam1_1.mat")
c1nonoise = c1["vidFrames1_1"]

c2 = scipy.io.loadmat("cam2_1.mat")
c2nonoise = c2["vidFrames2_1"]

c3 = scipy.io.loadmat("cam3_1.mat")
c3nonoise = c3["vidFrames3_1"]

height1, width1, channels, num_of_frames = c1nonoise.shape
height2,width2, channels, number_of_frames2 = c2nonoise.shape
height3,width3, channels, number_of_frames3 = c3nonoise.shape

"""# Load data with noise (Noisy Case)"""

c12 = scipy.io.loadmat("cam1_2.mat")
c1noise = c12["vidFrames1_2"]

c22 = scipy.io.loadmat("cam2_2.mat")
c2noise = c22["vidFrames2_2"]

c32 = scipy.io.loadmat("cam3_2.mat")
c3noise = c32["vidFrames3_2"]

height12, width12, channels, number_of_frames12 = c1noise.shape
height22,width22, channels, number_of_frames22 = c2noise.shape
height32,width32, channels, number_of_frames32 = c3noise.shape

def turn_to_gray(videodata):

  number_of_frames = videodata.shape[3]

  grey_frames = []

  # turn to greyscale
  for i in range(number_of_frames):

    grey = np.mean(videodata[:,:,:,i], axis=2)
    grey_frames.append(grey)

  grey_frames = np.array(grey_frames) # shape is (number of frames, height, width)

  return grey_frames

"""# Turn the data to gray"""

c1_gray = turn_to_gray(c1nonoise)
c2_gray = turn_to_gray(c2nonoise)
c3_gray = turn_to_gray(c3nonoise)
```

```
c1noisy_gray =turn_to_gray(c1noise)
c2noisy_gray =turn_to_gray(c2noise)
c3noisy_gray =turn_to_gray(c3noise)

"""The function takes input:
Reference can, gray video data, starth, endh, startw, endw.



Where starth, endh represents the reigon of the interest for the height (helps save computation t


If not specifying a specific region for height(example), input 0 for starth, Xgray.shape[1] for e

"""

def track(my_can, Xgray, starth, endh,startw, endw):

  # Define the coordinate arrays
  xcoordinate = []
  ycoordinate = []



  # Define height and width of the gray can
  # It'll also be the height and width of the regions we will be taking throughout the frames
  h = my_can.shape[0]
  w = my_can.shape[1]


  for frame in range(Xgray.shape[0]):
    value = 10000000

    for wid in range((endw - startw)-w):
      for heigh in range((endh - starth)-h):


        # Take a submatrix from the frame to test if it's your can
        test_can1 = Xgray[frame, starth + heigh : starth + h + heigh, startw + wid: startw + wid

        #calculate frob norm
        frob_norm = np.linalg.norm( my_can - test_can1 , "fro")

        if frob_norm <= value:

          value = frob_norm


          #save upper left coordinates
          xl = startw + wid
          yl = starth + heigh

          # Center
          x = (xl +xl +w)/2
```

```
            y = (yl +yl +h)/2

        xcoordinate.append(x)
        ycoordinate.append(y)


    return xcoordinate, ycoordinate

"""# XY Coordinate for C1 No Noise"""

# The can I'm using to compare
plt.imshow(c1nonoise[235:300,320:375,:,0])
plt.show()

my_can = c1_gray[0,235:300,320:375]

# ROI
starth1 = 50
endh1 = 435
startw1 = 300
endw1 = 389

# Track
x1coordinate, y1coordinate = track(my_can,c1_gray,starth1,endh1,startw1,endw1)

# Visualize the tracking
for i in range(c1nonoise.shape[3]):
  plt.imshow(c1nonoise[:,:,:,i])
  plt.scatter(x1coordinate[i],y1coordinate[i])
  plt.show()

# Save the coordinates
savetxt('xcords_fromgray_c1nonoise.csv', x1coordinate)
savetxt('ycords_fromgray_c1nonoise.csv', y1coordinate)

"""# C1 With noise """

startw12 = 300
endw12 = 450

# No specification on height
starth12 = 0
endh12 = c1noise.shape[0]

# Use the same reference can
my_can = c1_gray[0,235:300,320:375]


# Get track the can in noisy video
x1coordinate_noisy, y1coordinate_noisy = track(my_can,c1noisy_gray,starth12,endh12,startw12,endw1

# Save the coordinates
savetxt('noisy_xcoordinate1.csv', x1coordinate_noisy)
savetxt('noisy_ycoordinate1.csv', y1coordinate_noisy)
```

```
# Visualize the tracking
for i in range(c1noise.shape[3]):
  plt.imshow(c1noise[:,:,:,i])
  plt.scatter(x1coordinate_noisy[i],y1coordinate_noisy[i])
  plt.show()

"""# XY Coordinate for C2 with no noise"""

# Picture of the can that's being used as a reference
plt.imshow(c2nonoise[275:360,260:325,:,0])

my_can = c2_gray[0,275:360,260:325]

# Specify region of interest
starth2 = 60
endh2 = 425
startw2 = 240
endw2 = 360

# Track
x2coordinate, y2coordinate = track(my_can, c2_gray,starth2,endh2,startw2,endw2 )

# Visualize the tracking
for i in range(c2nonoise.shape[3]):

  plt.imshow(c2nonoise[:,:,:,i])
  plt.scatter(x2coordinate[i],y2coordinate[i])
  plt.show()

# Save the coordinates
savetxt('xcords_fromgray_c2nonoise.csv', x2coordinate)
savetxt('ycords_fromgray_c2nonoise.csv', y2coordinate)

"""# C2 with noise """

my_can = c2_gray[0,275:360,260:325]

# Specify the region of interest
starth22 = 0
endh22 = c2nonoise.shape[0]
startw22 = 150
endw22 = 500

# Track
noisy2_xcoordinate, noisy2_ycoordinate = track(my_can, c2noisy_gray,starth22, endh22,startw22,end

# Save the coordinates
savetxt('noisy2_xcoordinate.csv', noisy2_xcoordinate)
savetxt('noisy2_ycoordinate.csv', noisy2_ycoordinate)

# Visualize the tracking
for i in range(c2noise.shape[3]):
  plt.imshow(c2noise[:,:,:,i])
  plt.scatter(noisy2_xcoordinate[i],noisy2_ycoordinate[i])
  plt.show()
```

```
"""# XY Coordinate for C3 No noise """

# Picture of the can that I will be using to track the coordinates of the can in the frames
# Using frame 30 since it has a clear picture
# Also noticed that using smaller region gave more accurate results

plt.imshow(c3nonoise[30,300:320,410:470,:,30])
plt.show()

my_can = c3_gray[30,300:320,410:470]


# Reigon of interest
starth3 = 200
endh3 = 350
startw3 = 0
endw3 = c3nonoise.shape[1]

# Track
x3coordinate, y3coordinate = track(my_can,c3_gray,starth3,endh3,startw3,endw3)

# Save the coordinates
savetxt('xcords_fromgray_c3nonoise.csv', x3coordinate)
savetxt('ycords_fromgray_c3nonoise.csv', y3coordinate)

# Visualize the tracking
for i in range(c3nonoise.shape[3]):
  print(i)
  plt.imshow(c3nonoise[:,:,:,i])
  plt.scatter(x3coordinate[i],y3coordinate[i])
  plt.show()

"""# Camera 3 with noise"""

my_can = c3_gray[30,300:320,410:470]

# Reigon of interest starts and ends at
starth32 = 100
endh32 = 450
startw32 = 0
endw32 = c3noise.shape[3]

# Track
noisy3_xcoordinate, noisy3_ycoordinate = track(my_can,c3noisy_gray,starth32,endh32,startw32,endw3

# Save the coordinates
savetxt('noisy3_xcoordinate.csv', noisy3_xcoordinate)
savetxt('noisy3_ycoordinate.csv', noisy3_ycoordinate)

# Visualize the tracking
for i in range(c3noise.shape[3]):
  plt.imshow(c3noise[:,:,:,i])
  plt.scatter(noisy3_xcoordinate[i],noisy3_ycoordinate[i])
  plt.show()
```

```
"""# APPLY PCA"""

def get_data_matrix(x1,y1,x2,y2,x3,y3):

  # Get minimum number of frames (length of vectors correspond to number of frames of the videos)
  num_of_frames = np.min(np.array([x1.shape[0],x2.shape[0],x3.shape[0]]))

  # Create the matrix X
  X = np.vstack((x1,y1,x2[:num_of_frames],y2[:num_of_frames],x3[:num_of_frames],y3[:num_of_frames

  return X

def center_data(X):

  # Get the row means
  row_means = np.mean(X,axis=1)

  # Create the mean matrix
  Xbar = np.ones((X.shape[1],1))@np.reshape(row_means,(1,row_means.shape[0]))

  # Center data
  Xc = X-Xbar.T

  return Xc



"""# Ideal Case"""

# Create X and center data
X = get_data_matrix(x1coordinate,y1coordinate,x2coordinate,y2coordinate,x3coordinate,y3coordinate
Xc = center_data(X)

u,s,vt = np.linalg.svd(Xc)
v = vt.T
# Visualize the temporal evolutions of the first two observations
plt.plot(s[0]*v[:,0])
plt.plot(s[1]*v[:,1])
plt.legend(['$v_1(t)$', '$v_2(t)$'])
plt.xlabel("t")

# Truncate Data (keep until the first 150 cols of Xc)
Trunctuated_xc = Xc[:, :150]


# Compute SVD of truncated data
u_tr,s_tr,vt_tr = np.linalg.svd(Trunctuated_xc)


# Matrix V (whose cols are tne temporal evolutions)
v_tr = vt_tr.T

# Visualize the temporal evolutions of the truxated data
```

```
plt.plot(s_tr[0]*v_tr[:,0])
plt.plot(s_tr[1]*v_tr[:,1])
plt.legend(['$v_1(t)$', '$v_2(t)$'],loc=3)
plt.xlabel("t")

# Compute energy of each of the squared singular values
s_sq = s_tr**2
ratios_sq = np.array([i/np.sum(s_sq) for i in s_sq])

plt.scatter(range(1,7),ratios_sq)
plt.xlabel("i")
plt.ylabel("$\sigma_i^2/\sum_{i}^6 \sigma_i^2 $")
plt.show()

# Cumulative energy ratio of the first two singular values
ratios_sq[0]+ratios_sq[1]

"""# Project the data """

# Project trunctuated DATA

u2_tr = u_tr[:,:2]
Xp_trunc = u2_tr.T @Trunctuated_xc
Xp_trunc = Xp_trunc.T



"""# Apply the SINDy method """

Feature_Names = ["x","y"]
Optimizer = pysindy.STLSQ(threshold = 0.1)
model = pysindy.SINDy(feature_names = Feature_Names, optimizer = Optimizer)
model.fit(Xp_trunc, t = (11/226))
model.print()

# define the differential equations obtained from the SINDy method
from scipy.integrate import odeint
def RHS(z,t):
  a=-22.253
  b=-3.237
  c=32.032
  d=2.969
  dxdt = a +b*z[1]
  dydt = c +d*z[0]

  dzdt = [dxdt,dydt]
  return dzdt

# Intial conditions

z0 = [67.17865951,-66.79797315]

# Time points
t = np.arange(0,11,11/226)
```

```
#solve ode
z = odeint(RHS, z0, t)

# Visualize the results

plt.plot(t[:150],s_tr[0]*v_tr[:150,0])
plt.plot(t,z[:,0])
plt.xlabel("t")
plt.legend(['x(t)', 'Reconstructed x(t)'],loc='best')
plt.show()


plt.plot(t[:150],s_tr[1]*v_tr[:150,1])
plt.plot(t,z[:,1])
plt.xlabel("t")
plt.legend(['y(t)', 'Reconstructed y(t)'],loc='best')
plt.show()

"""# Noisy Case

"""

# Create X and center data

Xnoisy = get_data_matrix(x1coordinatenoisy,y1coordinatenoisy,x2coordinatenoisy,y2coordinatenoisy,
Xc_noisy = center_data(Xnoisy)

# SVD of Xc_noisy
u_noisy, s_noisy, vt_noisy = np.linalg.svd(Xc_noisy)

v_noisy = vt_noisy.T

# Compute energy of each squared singular value and plot it

s_noisy_sq = s_noisy**2
noisy_sum = np.sum(s_noisy_sq)
ratios_noisy=[]
for i in range(6):
  ratios_noisy.append(s_noisy_sq[i]/noisy_sum)

# Plot
plt.scatter(range(1,7),ratios_noisy)
plt.xlabel("i")
plt.ylabel("$\sigma_i^2/\sum_{i}^6 \sigma_i^2 $")
plt.show()

# Visualize the temporal evolutions
plt.plot(s_noisy[0]*vt_noisy[0,:])
plt.plot(s_noisy[0]*vt_noisy[1,:])
plt.legend(['$v_1(t)$', '$v_2(t)$'],loc=4)
plt.xlabel("t")
plt.show()

# Project data
u2_noisy = u_noisy[:,:2]
```

```
Xp_noisy = u2_noisy.T @ Xc_noisy
Xp_noisy= Xp_noisy.T

"""# Apply the SINDy method on the projected data """

Feature_Names2 = ["x","y"]
Optimizer2 = pysindy.STLSQ(threshold = 0.1)
noisy_model = pysindy.SINDy(feature_names = Feature_Names2, optimizer = Optimizer2)
noisy_model.fit(Xp_noisy, t =(12/314))
noisy_model.print()

# Integrate to and plot x(t) and y(t)

# define your function
from scipy.integrate import odeint
def RHS(z,t):
  a=-3.285
  b=-0.475
  c=-2.406
  d= 0.181

  dxdt = a +b*z[1]
  dydt = c +d*z[0]
  dzdt = [dxdt,dydt]
  return dzdt

# Intial conditions

z0 = [-7.19218182e+00,-2.35849157e+01]


# Time points
t = np.arange(0,12,12/314)

#solve ode
z = odeint(RHS, z0, t)

plt.plot(t,s_noisy[0]*v_noisy[:,0])
plt.plot(t,z[:,0])
plt.xlabel("t")
plt.legend(['x(t)', 'Reconstructed x(t)'],loc='best')
plt.show()
```